JAVA POWER TOOLS:
A FOUNDATION FOR INTERACTIVE HCI EXPLORATION

Viera K. Proulx, Richard Rasala, and Jeff Raab

College of Computer Science, Northeastern University, 360 Huntington Avenue, Boston, MA 02115 USA

ABSTRACT

In this paper we first discuss the key issues in GUI programming and identify abstractions that represent the basic GUI program building blocks. We then describe how the Java Power Tools (JPT) permit a GUI programmer to work at this higher level of abstraction. We present several examples of simple GUIs designed with the JPT to illustrate the use of the JPT idioms. Finally, we discuss the implications of using the JPT in computer science courses in three different contexts: as a tool for creating programs with GUIs, as a model for object oriented design and patterns, and as a tool for building interactive simulations of computer science concepts.

1. INTRODUCTION

The design of graphical user interfaces is difficult because the designer typically cannot test the design and explore different options until most of the application is implemented. The implementation is done either by direct programming (e.g. using Java with the Java Swing libraries) or by using visual tools. When using visual tools, the designer is limited by the options provided by another GUI designer and implementer: the one who designed the visual tools. On the other hand, the Java Swing libraries, while powerful, provide programming tools at a rather low level of abstraction so that the building of GUIs of any complexity requires tremendous effort. Because the code is so complicated, it is difficult for the designer to make changes and quickly explore different ways of organizing and presenting information to the user. In contrast, the Java Power Tools provide a robust environment for rapid GUI development in Java that encapsulates most of the tasks into single statements in a seamless manner. This permits a designer to work at the abstraction level of a visual tool while still having available the full programming strength and flexibility of Java.

2. GUI PROGRAMMING - THE KEY ISSUES

A program with a graphical user interface logically consists of two basic components: the internal part and the external part. The internal component contains the algorithms that accomplish the tasks of the program and uses necessary data structures to represent data models. The external component implements the user view of the system. It consists of various fields and widgets that may be used for user input of data, for display of some of the program results (views of the data model), and for controls through which the user can initiate actions.

There can be several ways in which a model and a view interact. In tight coupling, a change on one side of the model/view boundary should be reflected immediately by a corresponding change in the other side. In loose coupling, changes are coordinated only when certain user actions are initiated. It is important to realize that the two directions of coupling between the model and the view may be independent of each other. So, we may have a tight coupling in the direction from model to view in which every change in the model is immediately reflected in the view whereas a change in the view is not transmitted to the model until some relevant action is taken. For example, in an on-line form, the user may fill in several fields but nothing changes in the model until a Submit button is pressed. There may also be anticipatory coupling in which the model predicts the next user input and sets the view to this value (for example a counter for the next entry in a database). In this case, the view does not reflect the current state of the model but rather what the program anticipates it will become.

The key issues in the design of programs with GUI components are the following:

- representing the views used for input and output of information
- representing action controls and connecting them to the appropriate functions in the program
- representing the model in a way that enables communication with the view

- encapsulating of the communication between the model and the view

Conceptually, these issues are at the heart of a GUI programmer's thinking. The programming environment should make it possible to connect each GUI design concept to a single language command or at worst 2 or 3 language commands. The goal of the Java Power Tools (JPT) is to provide such a programming environment.

3. THE JPT DESIGN MODEL AND IMPLEMENTATION

The goal of JPT is to encapsulate the following basic GUI programming operations:

- create a model that can communicate with a view
- create a view that can communicate with a model
- extract a value from a view
- set the value of a view
- define the behavior of actions
- create controls that trigger such actions

3.1 Models and Views

The foundation for the communication between the data model and the view in JPT is the recognition of the fact that properly formatted String's can represent nearly all data models that correspond to user inputs. Thus, all views that intend to deliver information to a data model can implement an interface that guarantees that the view can set its display state from a String and can extract the information from its display state and deliver it in the form of a String. The two functions in the interface that perform these tasks are setViewState and getViewState. Similarly, all data models that expect to communicate with GUI views can implement an interface that guarantees that the data model can represent its internal state as a formatted String and set its internal state from a properly encoded String. The two functions in the interface that perform these tasks are toStringData and fromStringData. The following diagram illustrates the communication between a data model and its corresponding view:

```
Data model -> toStringData -> String -> setViewState   -> View
View       -> getViewState -> String -> fromStringData -> Data model
```

If there is a possibility that the input String may be malformed and an input error must to be caught before supplying the String to the data model, the view will implement the Fragile interface that activates a MalformedDataListener. Such views then provide automatic error dialogs and error handling strategies. The view will extract its data using one of two functions: requestObject or demandObject. If an error occurs, an error dialog will automatically be presented. In the case of requestObject, the user will be given the option of canceling the input operation in which case a CancelledException will be thrown so that the program will not proceed to act based on invalid input. In the case of demandObject, the user must supply valid data before the dialog releases control.

Finally, to make the whole JPT system scalable, the views and the data models are defined in a recursive manner. In other words, a data model can consist of several Stringable data models and can be itself contained in a larger Stringable data model. Similarly, several views can be composed into a larger view that may be itself contained in even larger view. The data model at the highest level can then extract the information from the entire compound view through a single operation or, vice versa, the compound view can be set in a single action using the model data..

3.2 Actions and Action Controls

The JPT also encapsulates the creation of button controls that trigger actions. The programmer need only define an appropriate action and install it in a special ActionsPanel that will then automatically create the button to perform this action and activate the necessary button listeners for the button. This encapsulation permits the programmer to focus on the algorithmics rather than on the low-level wiring of the button into the interface.

A special kind of control is a MouseActionAdapter that can be added to an appropriate view to track different mouse actions within the view. This adapter class handles the five events that Java names mouse events (press, release, click, enter, exit) and the two events Java names mouse motion events (move, drag). As before, the programmer merely needs to specify what action should be performed when the corresponding mouse behavior occurs within the view.

The JPT provides a special panel named BufferedPanel that is used to display graphics output. This panel is based on an underlying Java BufferedImage object that is used to save the graphics output so that window refresh is entirely automatic. This panel is set up so that it is centered in its enclosing frame and so that if this frame becomes smaller than the image area then the panel will automatically scroll. A BufferedPanel also comes equipped with a MouseActionAdapter whose default actions are null. By specifying a set of desired mouse actions for the seven mouse events, this panel may easily be configured for interactive graphics manipulation.

A similar approach is used with sliders. In the case of a slider only two events are important: sliding the knob and releasing the knob. We therefore configure a SliderView by specifying its sliding action and its release action.

In all cases, the common thread is that the programmer should specify what is conceptually important, that is, the actions, and all other technical details should be encapsulated and handled automatically.

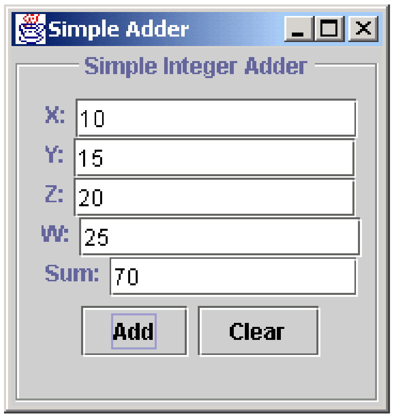
4. JPT IDIOMS AND EXAMPLES OF USE

In this section we will show some sample JPT code for building simple GUI's. We will show how each logical step in the GUI program design corresponds to one JPT Java statement. In general, to define and use a GUI widget or element, we must *construct* it, *place* it within a window, and, at an appropriate time, *extract* its user input for use by the internal model. The goal of the JPT is that these three logical steps should require exactly three statements and that all technical details should be encapsulated. Occasionally, when it is not convenient to specify all options in one parameter list, we may need one or two auxiliary statements solely for the purpose of selecting some variant or another. In particular, in the case of a TextFieldView in which the user input text may be in error, it will be necessary to specify certain aspects of the error recovery strategy using auxiliary statements.

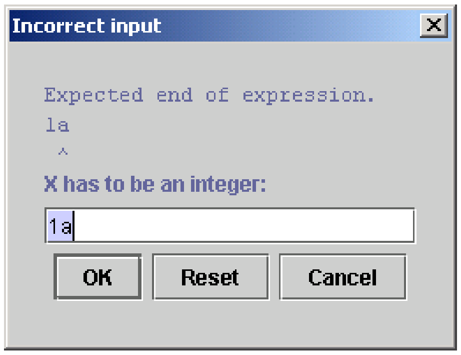
Because the JPT focuses on the three irreducible programming actions for widgets (construct, place, and extract), JPT code tends to follow common patterns that are similar for all widgets. The code is not obscured by a plethora of low level detail that varies from widget to widget. It is therefore possible to capture these patterns as programming idioms that may be used over and over in different combinations. As a result, when writing programs with JPT, the programmer may typically copy entire statements from an idiom collection and then modify the necessary fields. We will illustrate the use of these idioms on concrete examples.

4.1. The Simple Adder tutorial and the TextFieldView

The Simple Adder tutorial application shown at the left below uses four TextFieldView widgets for input of four integers and one TextFieldView widget for the display of the result of an addition operation. There are two action buttons, Add and Clear, that perform the addition of X, Y, Z, and W or that clear all text fields.



Error dialog for an input request without a suggestion



```
Expected end of expression.
1a
 ^
X has to be an integer:
1a
```

In the dialog box shown at the right, an input error has been detected in reading the text field for X. The error dialog title ("Incorrect input") and the prompt ("X has to be an integer:") are supplied in the construction of the text field view so that the view is prepared to handle errors automatically. The error message ("Expected end of expression.") is generated by the parser at the moment that the erroneous data ("1a") is extracted from the text field.

As examples, let us illustrate some of the different statements used in this tutorial to deal with a TextFieldView.

- To construct a text field view for an integer such as X, we write:

```
 protected TextFieldView xTFV =
   new TextFieldView(
      "0",                          // initial value displayed in xTFV
      "X has to be an integer:",    // prompt for correcting the input
      "Incorrect input");           // title for the error dialog box
```

  Notice that the constructor is passed three String's so that nothing in the constructor actually specifies that the value to be extracted has to be type int. This will be determined later when the value is extracted.

- To set the value in the summation text field view sumTFV to the result of the sum operation, we write:

```
sumTFV.setViewState(sum + "");
```

- To extract an integer value x from the text field view xTFV in order to perform the summation operation while also allowing the user to cancel during an error dialog box, we write:

```
try {
   x = xTFV.requestInt();        // requestInt calls fromStringData to parse
}
catch (CancelledException ex) {
   sumTFV.setViewState("");
   return;
}
```

  Should the user cancel the input in the error dialog, the variable x cannot be set so an exception is thrown and then caught to cause the program to clear the sum field and then return from the summation operation.

- To extract an integer value x from the text field view xTFV in order to perform the summation operation and to insist that a valid integer be provided by the user, we write:

```
x = xTFV.demandInt();           // demandInt calls fromStringData to parse
```

  For this form of input, no cancel button is provided in the error dialog box so the user must supply valid data.

- To set a suggested value to be offered in the error dialog box (for example "100"), we write:

```
XTFV.getInputProperties().setSuggestion("100");
```

  This will add a "Suggestion" button to the error dialog which when pressed will put "100" into the field.

The tutorial actually uses the 4 text fields to illustrate all combinations of demand or request and suggestion or no suggestion. In addition, the same power that we have illustrated for the integer type is available if a text field is used to enter the data for an object rather than a built-in type.

4.2. The Simple Adder tutorial and action buttons

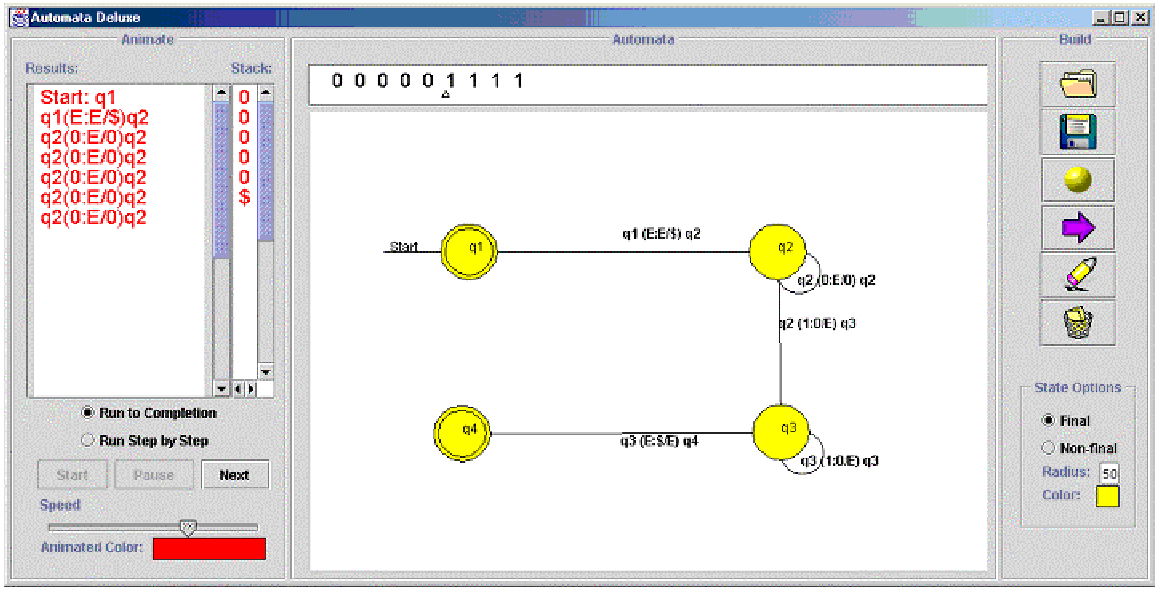
In order to add a button to the GUI that will invoke an action when selected, the programmer needs to specify four objects: an ActionsPanel object, an AbstractAction object that is instantiated by definition of its ActionPerformed method, an action member function of the enclosing class that will do the ActionPerformed, and a button label. The key step is illustrated by the following code:

```
actions.addAction(
   sum = new AbstractAction("Add") {     // Add is the button label
      public void ActionPerformed(ActionEvent evt) { sum(); }; } );
```

Notice that the sum action in the application is defined by an ActionPerformed method in the AbstractAction that calls the sum method in the application. This pattern is a standard idiom in the JPT.

5. THE AUTOMATA SIMULATION: A STUDENT PROJECT

It is clear from the previous section that the JPT can dramatically reduce the code needed for simple widgets. In fact, the JPT can be used to recursively build elaborate interfaces with the same simplicity. We will illustrate this with a screen snapshot from a simulation of push-down automata that was developed by our student, Jen McDonald, as an honors project. Similar simulations in other areas of computer science, mathematics, and science may be easily designed and constructed using the JPT.



Results:

```
Start: q1
q1(E:E/$)q2
q2(0:E/0)q2
q2(0:E/0)q2
q2(0:E/0)q2
q2(0:E/0)q2
q2(0:E/0)q2
```

Stack:

```
0
0
0
0
0
$
```

```
0 0 0 0 0 1 1 1 1
```

[✓ Run to Completion] Run Step by Step

[✓ Final] Non-final

Radius: 50

Color:

6. USES OF JPT IN EDUCATION

It is clear from the above examples that building programs with complex displays becomes quite straightforward when the JPT is used. As a result, it is easy to build freshman laboratory programming assignments where students are responsible for the algorithmics of some actions and for building the necessary data models that support the algorithmics while the GUI infrastructure is entirely provided. The visual feedback for the behavior of the program helps students understand the algorithms they are learning and also helps in debugging their solutions. As students become more experienced, they can gradually assume more and more of the GUI building on their own. Because the GUI programming is simple and follows intelligible patterns, students are empowered to explore design alternatives and not to simply accept the first GUI they happen to invent and program. Beyond the freshman year, the JPT can be used to encourage the creation of sophisticated GUIs in courses on software engineering and HCI and in courses that use simulation.

7. AVAILIBILITY OF THE JAVA POWER TOOLS AND RELATED PROGRAMS

The code for the Java Power Tools is open source with full JavaDoc documentation. These tools can be used not only to support GUI development at many levels of the curriculum but can also serve as a set of examples and case studies for courses in object-oriented design and patterns. The JPT and the associated curriculum materials, tutorials, demos, and references may be found at:

`http://www.ccs.neu.edu/teaching/EdGroup/JPT/`

The authors wish to acknowledge the support of Northeastern University and the National Science Foundation (Grant DUE 9950829) for the development of the Java Power Tools and the curriculum materials.

[1] Rasala, R., Raab, J., Proulx, V. K., *Java Power Tools: Model Software for Teaching Object-Oriented Design*, SIGCSE Bulletin, 2001, Vol. 33, No. 1, 297-301.